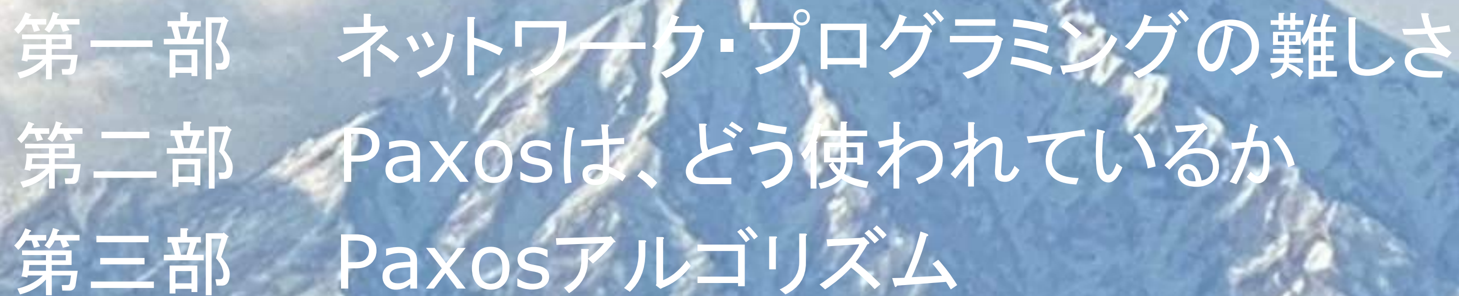
分散合意アルゴリズム -- Paxos



分散合意アルゴリズム – Paxos

Agenda

分散合意アルゴリズム – Paxos

はじめに　なぜ、今、Paxosなのか？

第一部　ネットワーク・プログラミングの難しさ

1. 分散コンピューティングについての考察
2. ネットワークの遅延について
3. システムの進化とその複雑さについて

第二部　Paxosは、どう使われているか

1. Paxosの働きをイメージする
2. Paxosとコンテナー技術
3. LeaseとPaxos

第三部　Paxosアルゴリズム

1. たとえ話で理解するPaxos (1)
2. たとえ話で理解するPaxos (2)
3. Paxos アルゴリズム (1)
4. Paxos アルゴリズム (2)

はじめに
なぜ、今、Paxosなのか？

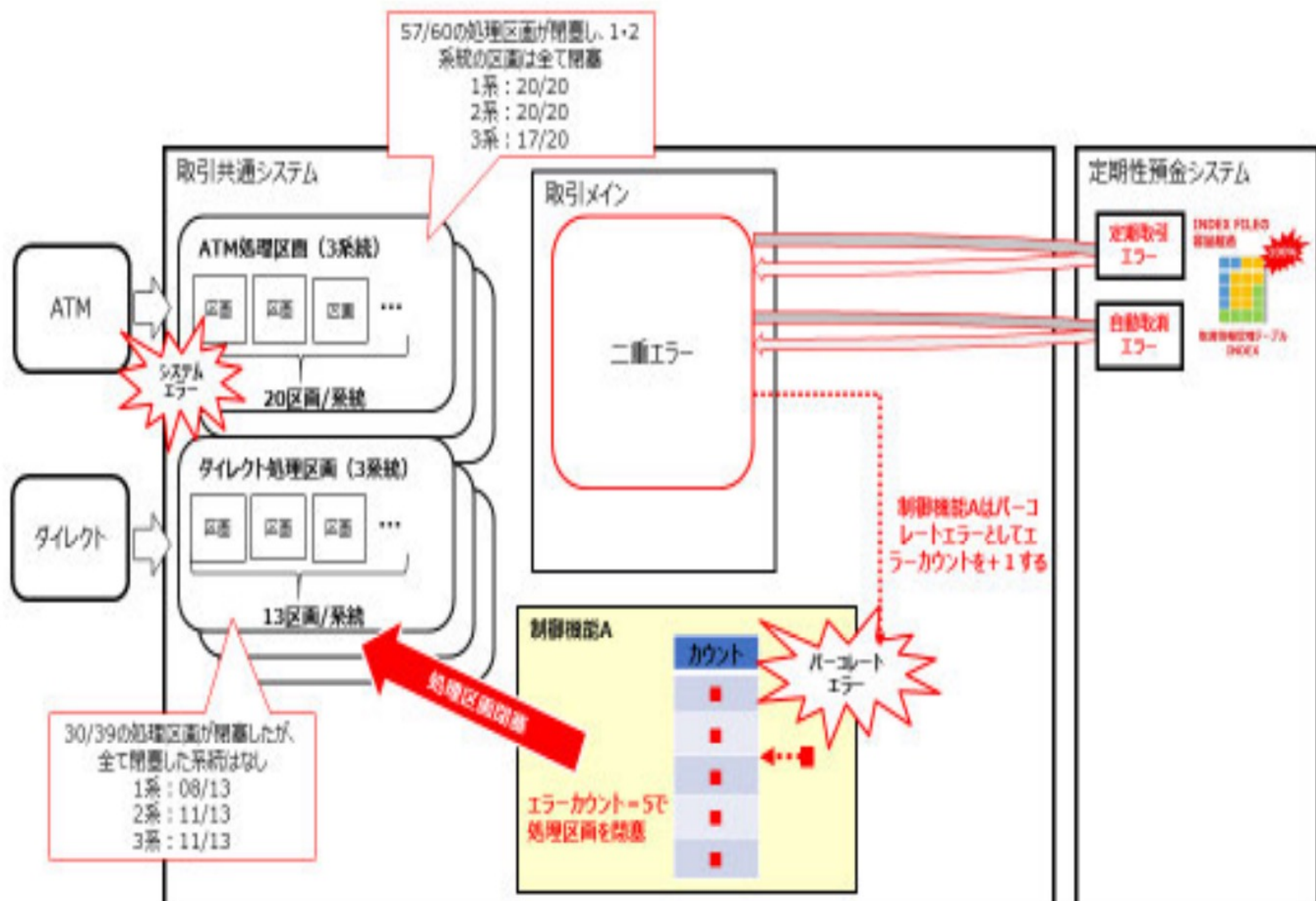
なぜ、いま、Paxosなのか？

今回のマルレクのテーマは、代表的な「分散合意アルゴリズム」であるPaxosの紹介です。
https://paxos.peatix.com/

なんで今頃Paxosなのかと感じる人もいると思います。IT系の話題を取り上げるのは久しぶりなので、最初に、このセミナーの背景を述べておこうと思います。

それは、現在の日本で、大規模なシステム障害が連続して起きていることと、すこし関係しています。

【図 14】



■ 参考2. データアクセス障害の発生原因

■故障発生時の状況

各種システム

自治体様業務システム

Jip−Base

仮想OS

物理サーバ

ディスクアクセス

ストレージコントローラ1

ストレージコントローラ2

共有ディスク

仮想ディスク（OS/データ）

ストレージ

②アクセスできない仮想ディスクを使用している仮想OSが利用不可

①ストレージの物理的回復後も、論理的な領域に不整合が発生した結果、一部仮想ディスクに対して、正しくアクセスできない事象が発生

https://www.jip.co.jp/news/pdf/20191216.pdf

21世紀初頭のIT技術の変化
クラウドとクラウド・デバイスの登場

21世紀初頭のIT技術とITビジネスの大きな変化は、膨大な計算資源を集中してネットワークを通じてサービスを提供するクラウドと呼ばれる巨大なシステムの成立によって特徴づけられます。

こうしたクラウドへのリソースの集中に対応して、クラウドのサービスを利用するクラウド・デバイスとしてのスマートフォンが拡散し、ほとんど全人類に普及します。

技術者のライフサイクルからみれば、このクラウド時代への移行は、驚くほど短期間に急速に進行しました。

ネットワーク・プログラミングの時代

大規模システムというのは、単に、システムの規模が大きいことを意味しません。今日では、大規模システムは、設計者がそれを意識するか否かに関わらず、クラウドと同じようにネットワークをまたいだ大規模分散システムの形を取ります。

プログラミングのスタイルも大きく変わりました。サーバーサイドでサービスを作るプログラマも、クライアントサイドでアプリを作るプログラマも、本人がそれを意識すると否とに関わらず、今日のプログラマの大部分は、ネットワーク・プログラマです。

クラウド技術の特徴
ScalabilityとAvailability

クラウド技術はもちろんネットワーク技術なのですが、クラウド技術成立の前提には、「ネットワーク上で規模を拡大しながら、システム全体としては障害を起こさないシステムを作ることは難しい」という認識があったことです。この認識は「ハードウェアの障害は起きるものだ」という障害観の成立とともに、重要なものだと思います。

クラウドの技術は、まさにこうした困難を乗り越える技術として登場しました。また、それは、冒頭でも見たように、大きな成功を収めました。

IT技術者の持つべきスキル

ただ、僕は、変化のスピードが早すぎて --- 繰り返しますが、技術者のライフサイクルから見ればということですが --- ネットワーク上でシステムを構築するというクラウド時代の中心的変化の技術的な意味が、クラウド以前の技術者にもクラウド以降の技術者にも、きちんと伝わっていないのではという心配をもっています。

それは、クラウド以前の技術者だけの問題ではありません。若い技術者は、必ずしもクラウド技術の一時代を画したGoogleの "Google File System (GFS)" や"BigTable" に関心を持っているわけではないように思えます。

基本を学ぼう

GFSとBigTableは、直接的には異なる目的のために開発されたものです。ただ、両者で共通に利用されている "Chubby"と呼ばれた「分散ロック・サービス」は、Paxosの実装にほかなりません。見かけは地味ですが、こうした形での「分散合意アルゴリズム」の利用には、クラウド技術のエッセンスが詰まっていると僕は考えています。

10年以上前の技術の昔話をするつもりはありません。「風が吹けばオケ屋が儲かる」式に「Paxosがわかればクラウド技術がわかる」という話をするつもりもありません。ただ、知らないままに通り過ぎる訳にはいかない技術は存在します。

その上、「分散合意」技術、けっして完成した過去の枯れた技術ではありません。その重要性は、現代のIT技術においても、新しい装いのもとに引きつがれています。

セミナーの構成

今回のセミナーは、次のような構成を考えています。

第一部　ネットワーク・プログラミングの難しさ
第二部　Paxosは、どう使われているか
第三部　Paxosアルゴリズム

第一部
ネットワーク・プログラミングの難しさ

第一部の目標と参考文献

第一部では、「ローカルなプログラミングとリモートなプログラミングをはっきりと区別すべき」だという立場を明確にしたJim Waldoの洞察に依拠して、ネットワーク・プログラミングの「難しさ」を振り返ってみようと思います。

- Jim Waldo
  “A Note on Distributed Computing”
  http://www.sunlabs.com/techrep/1994/smli_tr-94-29.pdf
- “Complexity Quanta and Platform Definition”
  https://wstrange.wordpress.com/2006/10/05/summary-jim-waldos-keynote-at-the-10th-jini-community-meeting/

第一部

ネットワーク･プログラミングの難しさ

1. 分散コンピューティングについての考察
2. ネットワークの遅延について
3. システムの進化とその複雑さについて

分散コンピューティングについての考察

A Note on Distributed Computing

Jim Waldo
Geoff Wyant
Ann Wollrath
Sam Kendall

SMLI TR-94-29 November 1994

https://dl.acm.org/doi/book/10.5555/974938

Abstract

我々は、分散システムで相互作用するオブジェクトは、単一のアドレス空間で相互作用するオブジェクトとは本質的に異なる方法で処理される必要があると主張する。

分散システムでは、プログラマはレイテンシーを意識し、異なるメモリアクセスのモデルを持ち、同時実行や部分的な障害の問題を考慮する必要があるため、このような違いが必要となる。

本研究では、ローカルオブジェクトとリモートオブジェクトの区別を克服しようとした多くの分散システムを見て、そのようなシステムが堅牢性と信頼性の基本的な要件をサポートしていないことを示す。

このような失敗は、これまで構築されてきた分散システムの規模が小さいために隠蔽されてきた。しかし、近い将来に予想される企業規模の分散システムでは、そのような隠蔽は不可能である。

最後に、もし、分散システムに真剣に取り組もうとするなら、システムレベルとアプリケーションレベルの両方のプログラマーやデザイナーに何が求められるのかを議論して終わる。

Introduction

現在の分散オブジェクト指向システムの多くは、そのオブジェクトが、存在論的に一つのクラスを形成していると言う仮定に基づいている。

このクラスは、オブジェクトがサポートする一連のインターフェースの仕様と、それらのインターフェースにおける操作のセマンティクスによって完全に記述できるすべてのエンティティから構成されている。

このクラスには、単一のアドレス空間を共有するオブジェクト、同じマシン上の別々のアドレス空間にあるオブジェクト、異なるマシン上の別々のアドレス空間にあるオブジェクト（おそらく異なるアーキテクチャを持つ）が含まれる。

すべてのオブジェクトが本質的に同じ種類のエンティティであるという考え方によれば、相対的な位置の違いは、オブジェクトの実装の側面に過ぎない。実際、オブジェクトの位置は、オブジェクトがあるマシンから別のマシンに移動したり、オブジェクトの実装が変更されたりすることで、時間とともに変化する可能性がある。

このようなオブジェクトの統一的な見方は間違っているというのが、このノートの主題である。

分散オブジェクトの相互作用と非分散オブジェクトの相互作用には基本的な違いがある。この違いを無視したモデルに基づいて分散オブジェクト指向システムを構築することは、失敗する運命にあり、分散オブジェクト指向システムという概念を業界全体で否定することになりかねない。

The Vision of Unified Objects

プログラマの視点では、アドレス空間を共有するオブジェクトと、異なる大陸にある異なるアーキテクチャの2台のマシン上にあるオブジェクトとを本質的に区別しないという考え方がある。

このビジョンは、リモート・プロシージャ・コール（RPC）システムの目標を、オブジェクト指向のパラダイムに拡張したものと考えることができる。RPCシステムは、アドレス空間をまたぐ関数呼び出しを、（クライアントプログラムからは）ローカル関数呼び出しのように見せようとするものである。

実際には、ローカルメンバー関数の呼び出しと大陸横断的なオブジェクトの呼び出しは、もちろん同じではない。

このビジョンの概念的な正当性の1つは、呼び出しがローカルであるかリモートであるかは、プログラムの正しさに影響しないということである。

このようなシステムには多くの利点がある。

それは、ソフトウェアメンテナンスのタスクを根本的に変えることができることである。

オブジェクト間のインターフェースが一定であれば、そのオブジェクトの実装を自由に変更することができる。リモートサービスをアドレス空間に移動させたり、アドレス空間を共有しているオブジェクトをローカルな要求やニーズに応じて分割して別のマシンに移動させたりすることができる。

このビジョンは、一見するともっともらしい次のような原則を中心に据えている。

- アプリケーションがどのような状況で展開されるかに関わらず、特定のアプリケーションに対する単一の自然なオブジェクト指向設計がある。
- 故障やパフォーマンスの問題は、アプリケーションを構成するコンポーネントの実装に関連しており、初期設計ではこれらの問題を考慮する必要はない。
- オブジェクトのインターフェースは、そのオブジェクトが使用されるコンテキストとは独立している。

残念ながら、これらの原則はすべて間違っている 。

Deja Vu All Over Again

分散コンピューティングの難しい問題は、物事をどのようにワイヤーに乗せたり外したりするかという問題ではない。

分散コンピューティングの難しい問題は、部分的な障害への対処や、中央のリソース・マネージャが欠けていることである。

分散コンピューティングの難しい問題は、適切なパフォーマンスを保証し、同時実行の問題に対処することである。

ハード面では、ローカルとディストリビュートの間のメモリアクセスパラダイムの違いが問題となる。

分散アプリケーションを書こうとする人は、通信プロトコル・プログラミング・インターフェースではなく、これらの分野にすべての努力を費やしていることにすぐに気づく。

ネットワークの遅延について

“Latency as an Effect”

“Principles of Reactive Programming”
Coursera lecture by **Erik Meijer**
https://class.coursera.org/reactive-001/lecture/51

2013 Nov 4th
"Principles of Reactive Programming"

Martin Odersky
École Polytechnique
Fédérale de Lausanne

Erik Meijer
Applied Duality

Roland Kuhn
Typesafe Inc.

https://www.coursera.org/course/reactive

```
trait Socket {
  def readFromMemory(): Array[Byte]
  def sendToEurope(packet: Array[Byte]): Array[Byte]
}

val socket = Socket()
val packet = socket.readFromMemory()
val confirmation = socket.sendToEurope(packet)
```

PCでの処理時間

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

http://norvig.com/21-days.html#answers

シーケンシャルな実行時間

```
val socket = Socket()
val packet = socket.readFromMemory()
  //  50,000 ナノ秒ブロックする
  //  例外が発生しない時には、次の処理へ
val confirmation = socket.sendToEurope(packet)
  //  150,000,000 ナノ秒ブロックする
  //  例外が発生しない時には、次の処理へ
```

1 ナノ秒を 1 秒だと思うと

execute typical instruction	1 second
fetch from L1 cache memory	0.5 seconds
branch misprediction	5 seconds
fetch from L2 cache memory	7 seconds
Mutex lock/unlock	½ minute
fetch from main memory	1½ minutes
send 2K bytes over 1Gbps network	5½ hours
read 1MB sequentially from memory	3 days
fetch from new disk location (seek)	13 weeks
read 1MB sequentially from disk	6½ months
send packet US to Europe and back	5 years

1ナノ秒を1秒だとした場合の実行時間

```
val socket = Socket()
val packet = socket.readFromMemory()
  //  3日間ブロックする
  //  例外が発生しない時には、次の処理へ
val confirmation = socket.sendToEurope(packet)
  //  5年間ブロックする
  //  例外が発生しない時には、次の処理へ
```

12 months to walk coast-to-coast
3 months to swim across the Atlantic
3 months to swim back
12 months to walk back

Humans are twice as fast as computers!

システムの進化とその複雑さについて

Complexity Quanta and Platform Definition

Jim Waldo
Distinguished Engineer
Sun Microsystems Laboratories

goo.gl/tw9F8c
Summary of Jim Waldo's Keynote
goo.gl/DBtmv5

複雑さにおける質的な飛躍

- **線形実行 (SEQ)** – 人生は善良でシンプルであった
- **マルチ･スレッド (MT)** – ツールと優秀なプログラマがMTについて考えることが必要
- **マルチ･プロセス (MP)** – カーネルの開発者だけでなく誰もが利用できる。実際には、MTの前に起きた。
- **マルチ･マシン (MM)** 同一ネットワーク上の – マルチ･プロセスと同じではないのだが、ある人たちは、そう考えている
- **信頼できないマルチ･マシンたち(MMU)** – 本質的には、Webの世界である

それぞれの段階を通り抜ける際、我々は、何かを失う

- マルチ・スレッドへ：
  我々は、順序を失う（複数のことが同時に起こる）。これは、難しい。なぜなら、我々は、自然には、シーケンシャルに考えるから。

- マルチ・プロセスへ：
  単一のコンテキスト（すなわち、我々が信頼しうる共有コンテキスト）を失う。グローバルな状態が、開発のあらゆるところで利用される。（すべてをスタティックに考えよ）

それぞれの段階を通り抜ける際、我々は、何かを失う

- マルチ・プロセスからマルチ・マシンへ：
  我々は、状態を失う。「システム」のグローバルな状態というのは、虚構である。興味深い分散システムには、整合的な状態というものは存在しない。（Lamport:http://research.microsoft.com/users/lamport/pubs/pubs.html ）
  分散OSのプロジェクトは、グローバルな状態を導入しようとしたが、大々的に失敗した。

- 信頼できないマルチ・マシンたちへ：
  誰を信ずることが出来るか分からない難しい状況の中で、我々は信頼を失う。

しかし、我々は何かを得てきた

- Seq to MT：パラレル処理
- MT to MP：プロセスの分離(安全を与える)
- MP to MM：独立した失敗 (何かまずいことが起きても、システムの部分は生きのこる)
- MM to MMU：スケール (webスケール、インターネットスケール). 誰か他の人のリソースを利用せよ（あるいは、他の誰かが、我々のリソースを利用することを認めよ）

MP->MM で発生する Partial Failureへの対応としての
Leaseのアイデアは、Waldoの大きな仕事である。

第二部

Paxosは、どう使われているか

第二部の目標と参考文献

第二部の目標は、Paxosがさまざまの大規模分散システムで利用されていることを知り、それがどのような役割を果たしているかのイメージを持つことです。

- ”The Chubby lock service for loosely-coupled distributed systems”
  https://static.googleusercontent.com/media/research.google.com/ja//archive/chubby-osdi06.pdf
- ”Large-scale cluster management at Google with Borg”
  https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43438.pdf

第二部

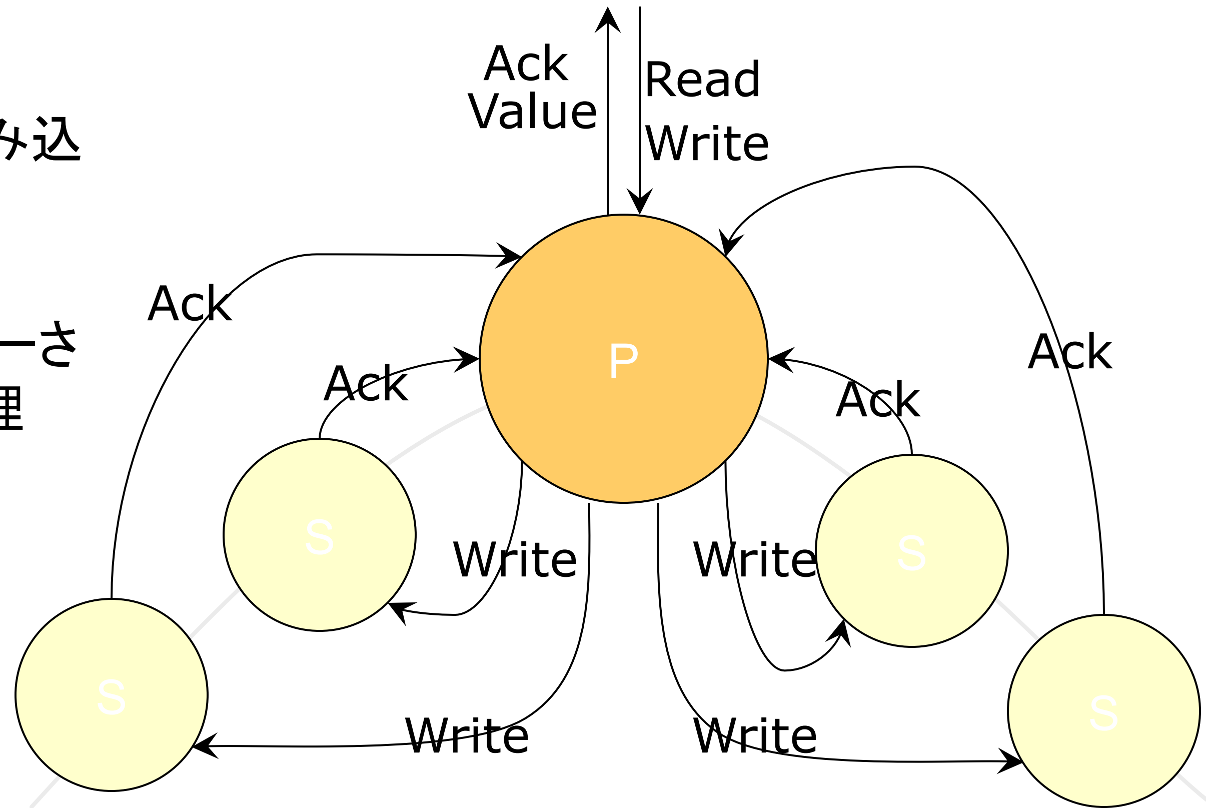
Paxosは、どう使われているか

1. Paxosの働きをイメージする
2. Paxosとコンテナー技術
3. LeaseとPaxos

Paxosの働きをイメージする

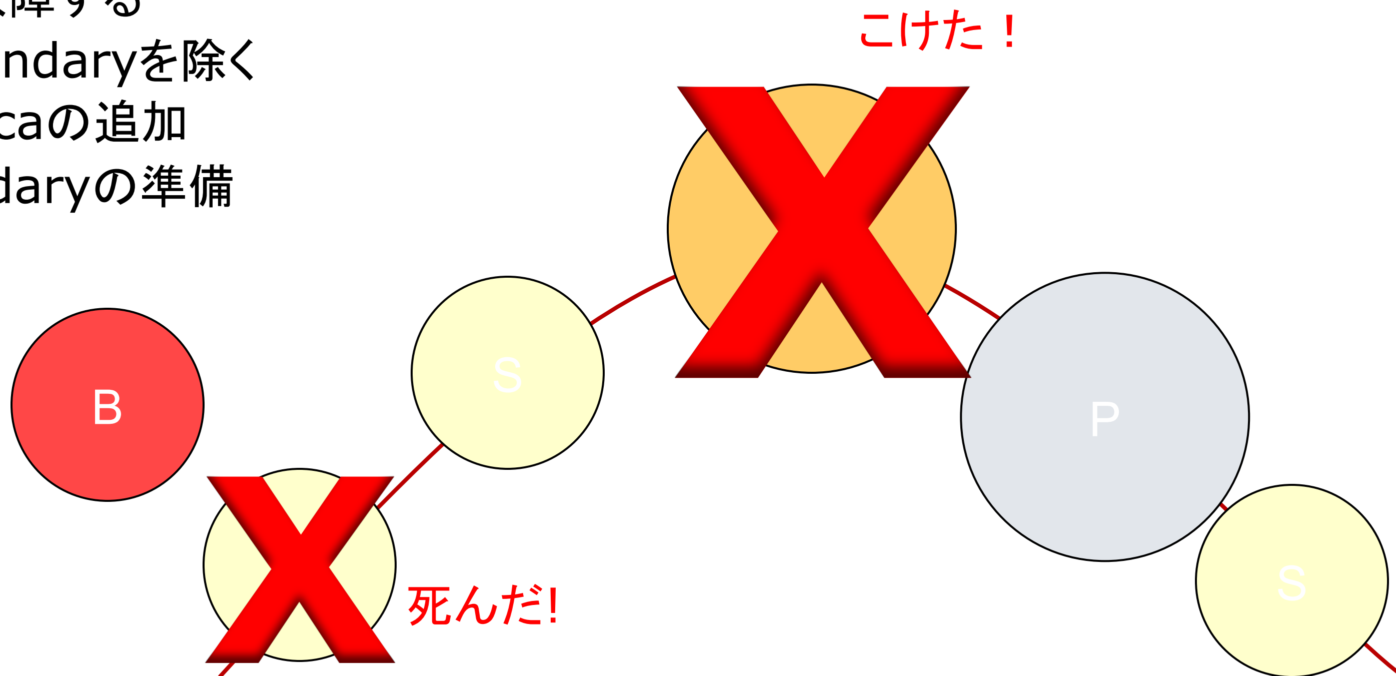
Azureのデータノードの多重化

- データの読み込みはPrimaryノードからの読み込みで完了する
- データの書き出しはSecondaryノードにコピーされる。この際、多数決原理(quorum)に従う。



Azureのデータノードの再構成

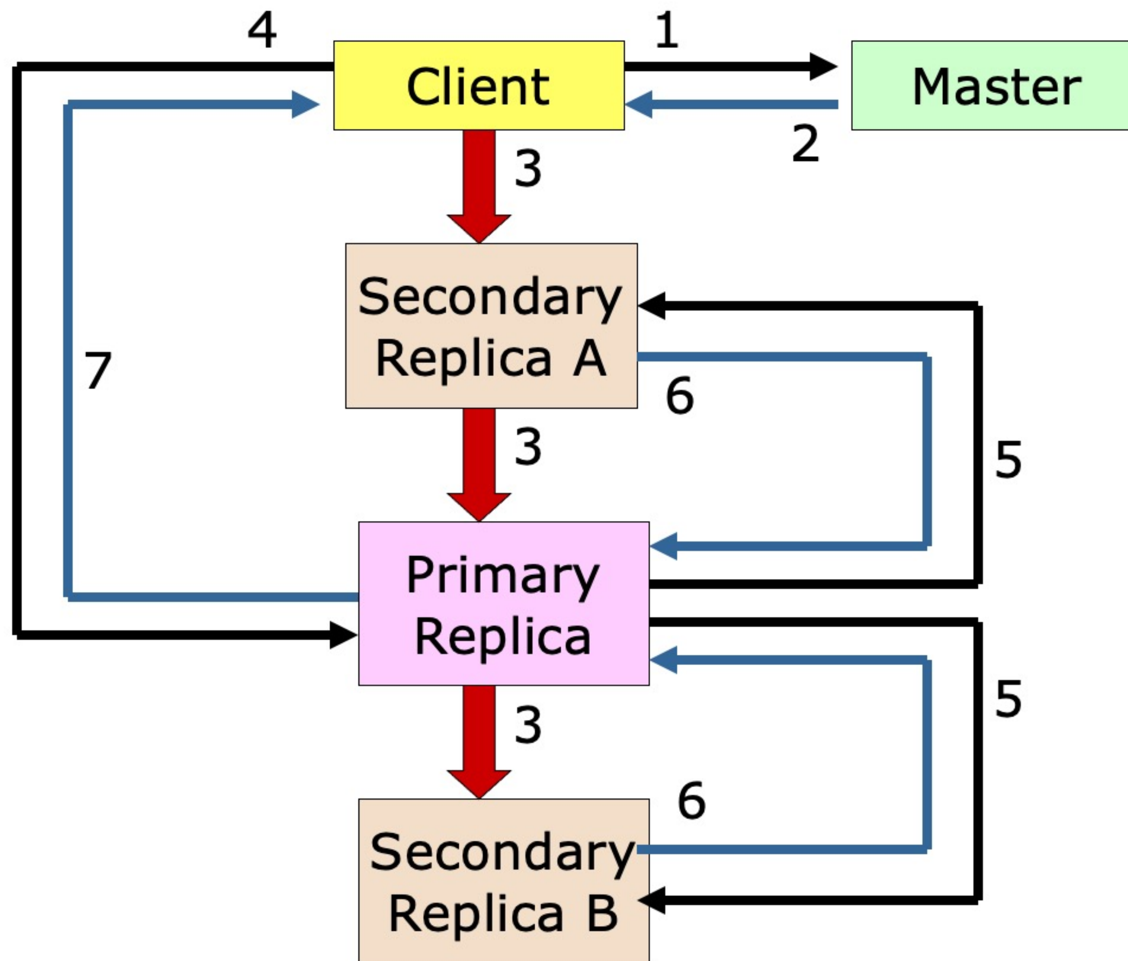
- 再構成のいくつかのタイプ
  - Primary が故障する
  - 故障したSecondaryを除く
  - 修復したreplicaの追加
  - 新しいSecondaryの準備



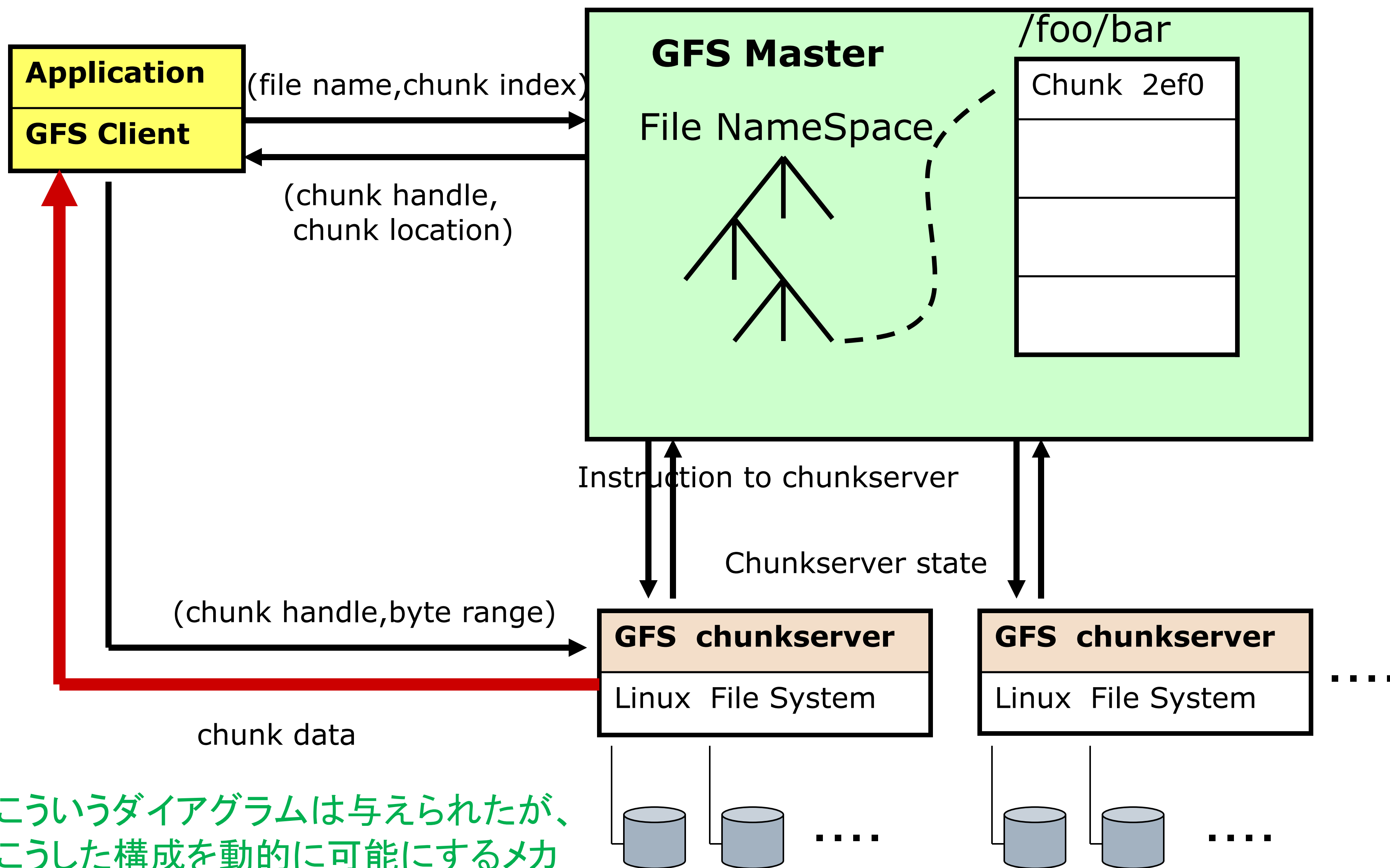
- 前提
  - 故障の検出
  - リーダーの選挙

Paxosとコンテナー技術

クラウドは、ノードの多重化のシステム？



GFS Architecture



こういうダイアグラムは与えられたが、
こうした構成を動的に可能にするメカ
ニズムは伏せられたままだった

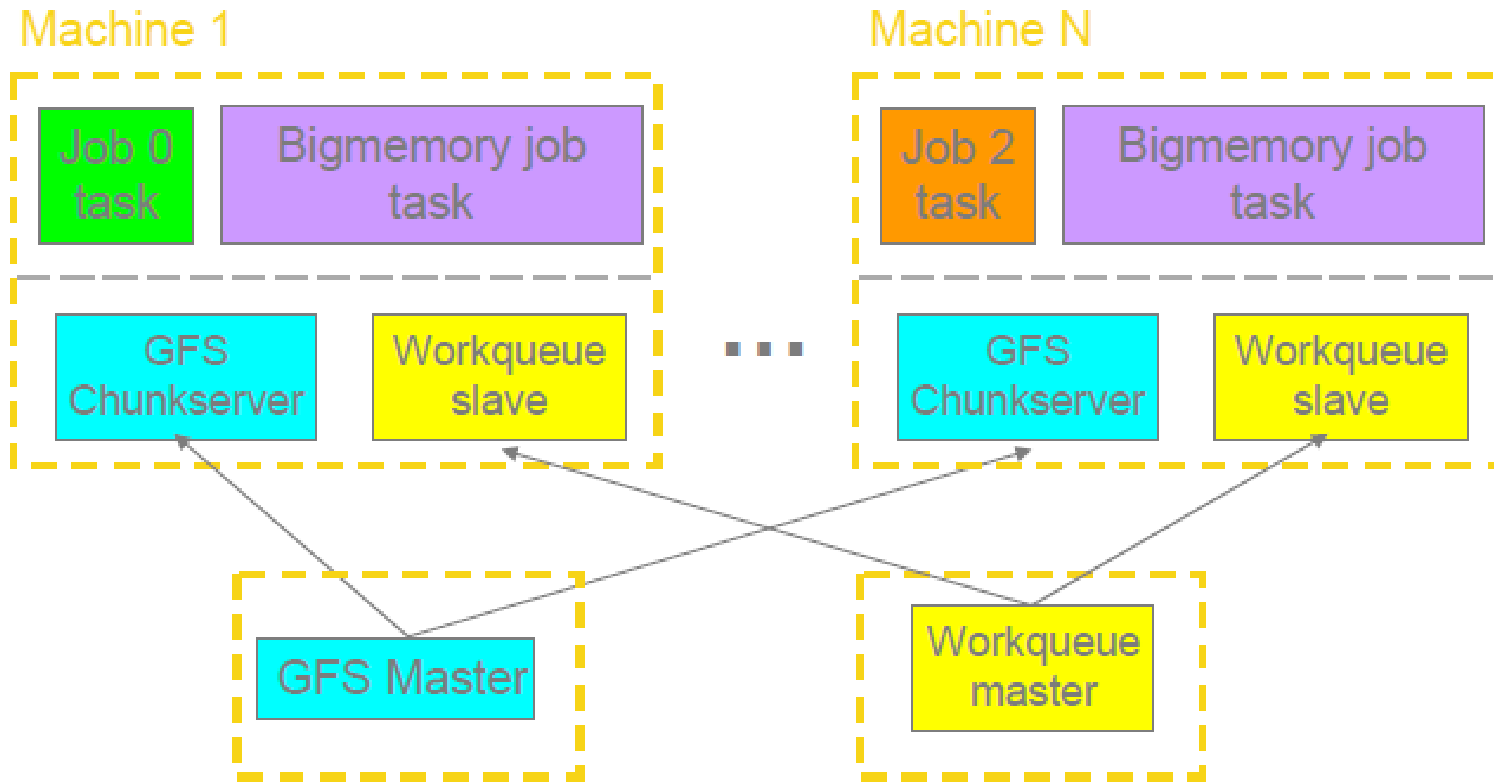
クラウドを支える技術

「EC2/S3のようなサービス提供ができるためには、ネットワーク上に分散して存在している物理的なディスクや物理的なサーバを、仮想化して論理的に管理して、稼働していないものはリソース・プールに登録して、変動する要求に応じて動的にそこからリソースを取り出して仕事を割り当てて、スケーラブルなサービス提供を保障しなければならない。AmazonのEC2/S3のサービス開始は、クラウドを支えるこうした課題の重要性に、皆の眼を向けさせた。」

「クラウドの成立過程とその技術的特徴について」
情報処理 Vol.50 No.11 Nov. 2009
https://goo.gl/SdxUjf

しかし、その実装は不明なままだった

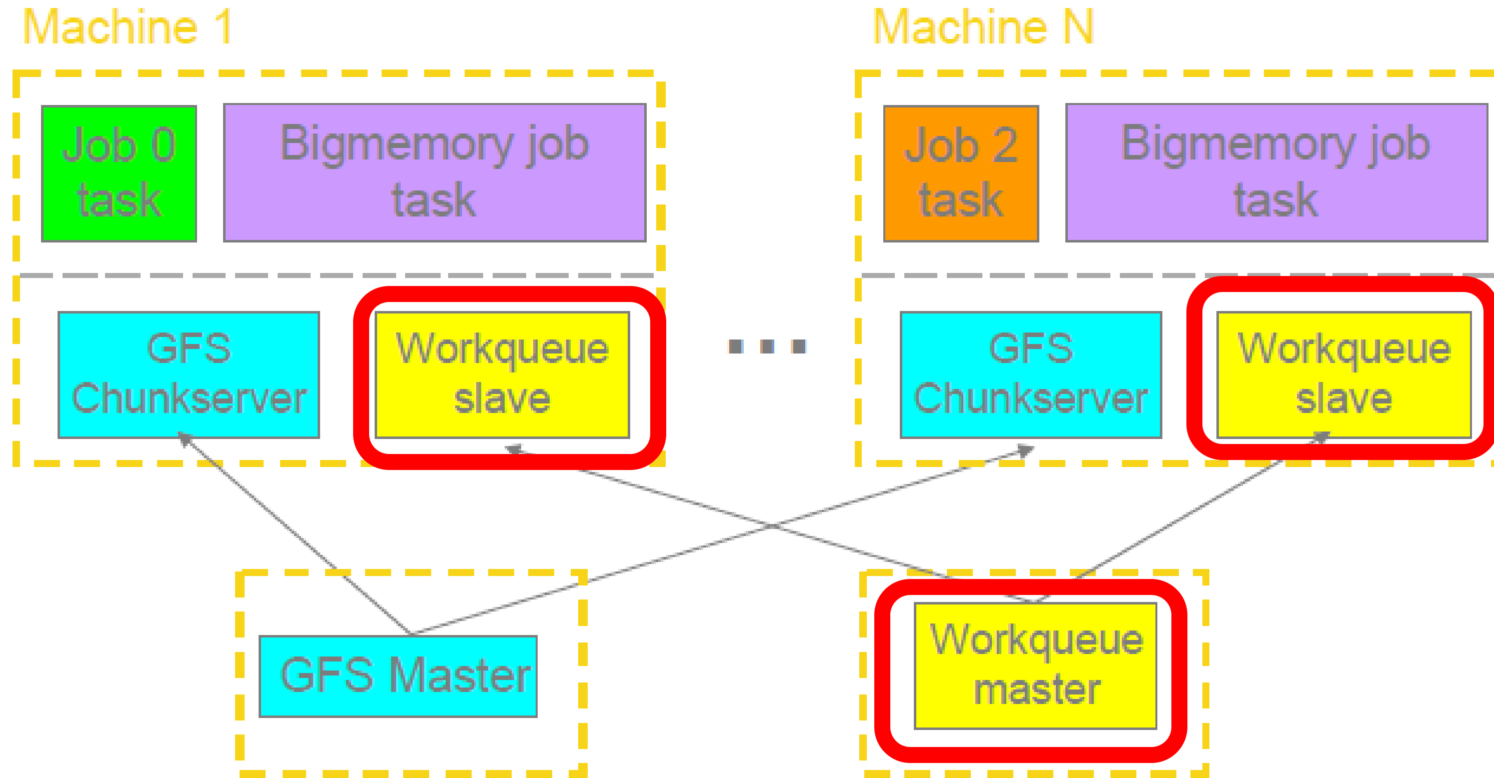
Basic Computing Cluster



ある時、ある図に見慣れないものが書かれてあった。

Google

Basic Computing Cluster



ある時、ある図に見慣れないものが書かれてあった。
Workqueueって何？　多分、clusterの管理をして
いるのはこれだと思ったが、どこを探しても説明はなかった

Google

Google, Borgの情報公開

2015年4月、Googleは、Borgの情報を初めて公開した。
“Large-scale cluster management at Google with Borg” https://goo.gl/aN03bI

この発表に対して、Auroraのアーキテクトのコメントを含む記事が、出ている。“Google Lifts the Veil on Borg, Revealing Apache Aurora’s Heritage”http://goo.gl/Nv8ZIQ

ようやく情報が出た。10年以上、隠していたことになる。

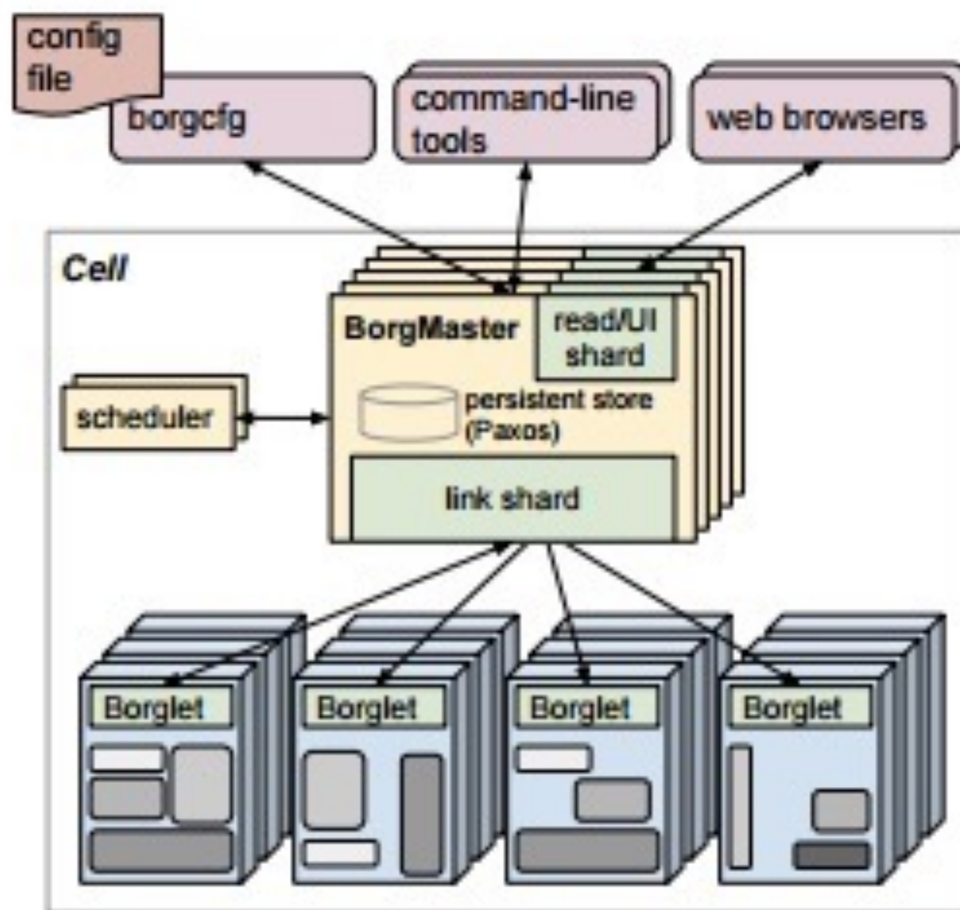
「Googleの秘密兵器」

Borgは、以前から、その存在は知られていたが、GFS, MapReduce, BigTable とは異なって、その技術的詳細をGoogleが明かすことはなかった。

Borgは、Googleの大規模分散の中核技術であり、ある人は、それを「Googleの秘密兵器」「Googleの急速な進化の、もっともよく保たれた秘密」と呼んでいた。

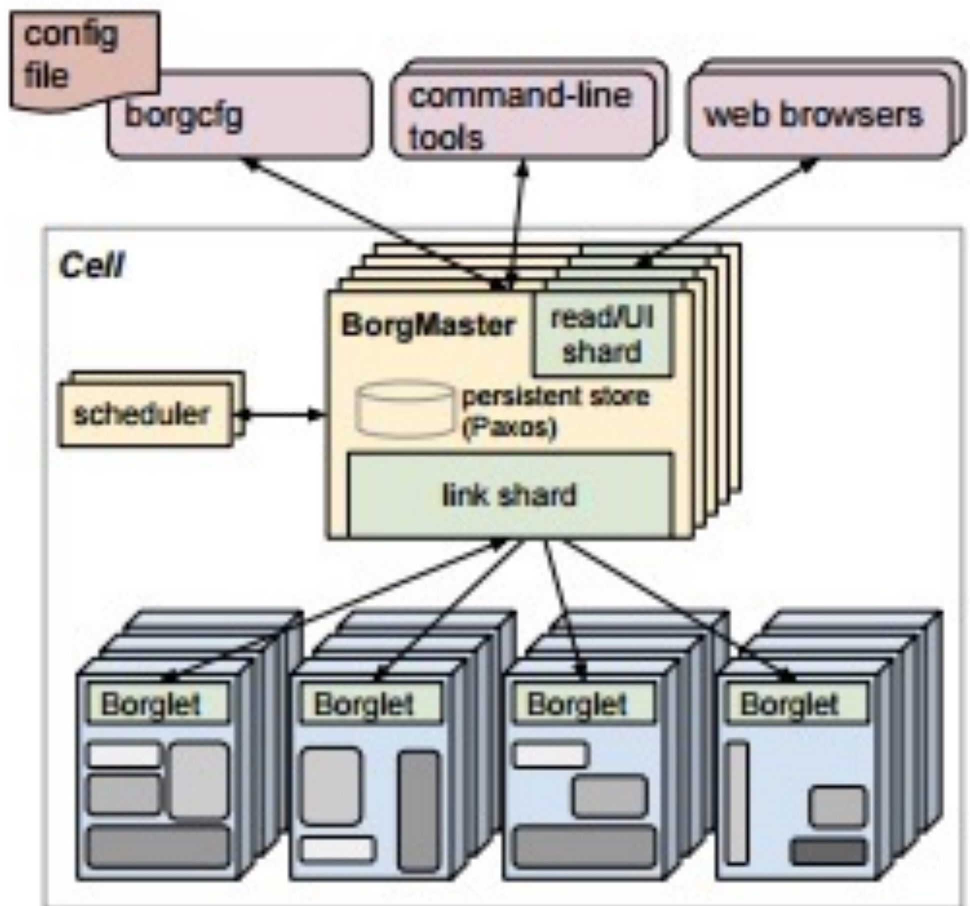
- “Return of the Borg: How Twitter Rebuilt Google�‘s Secret Weapon”http://goo.gl/QyhGjx
- "Twitter’s Aurora and How it Relates to Google’s Borg (Part 1)"http://goo.gl/BRhL7x

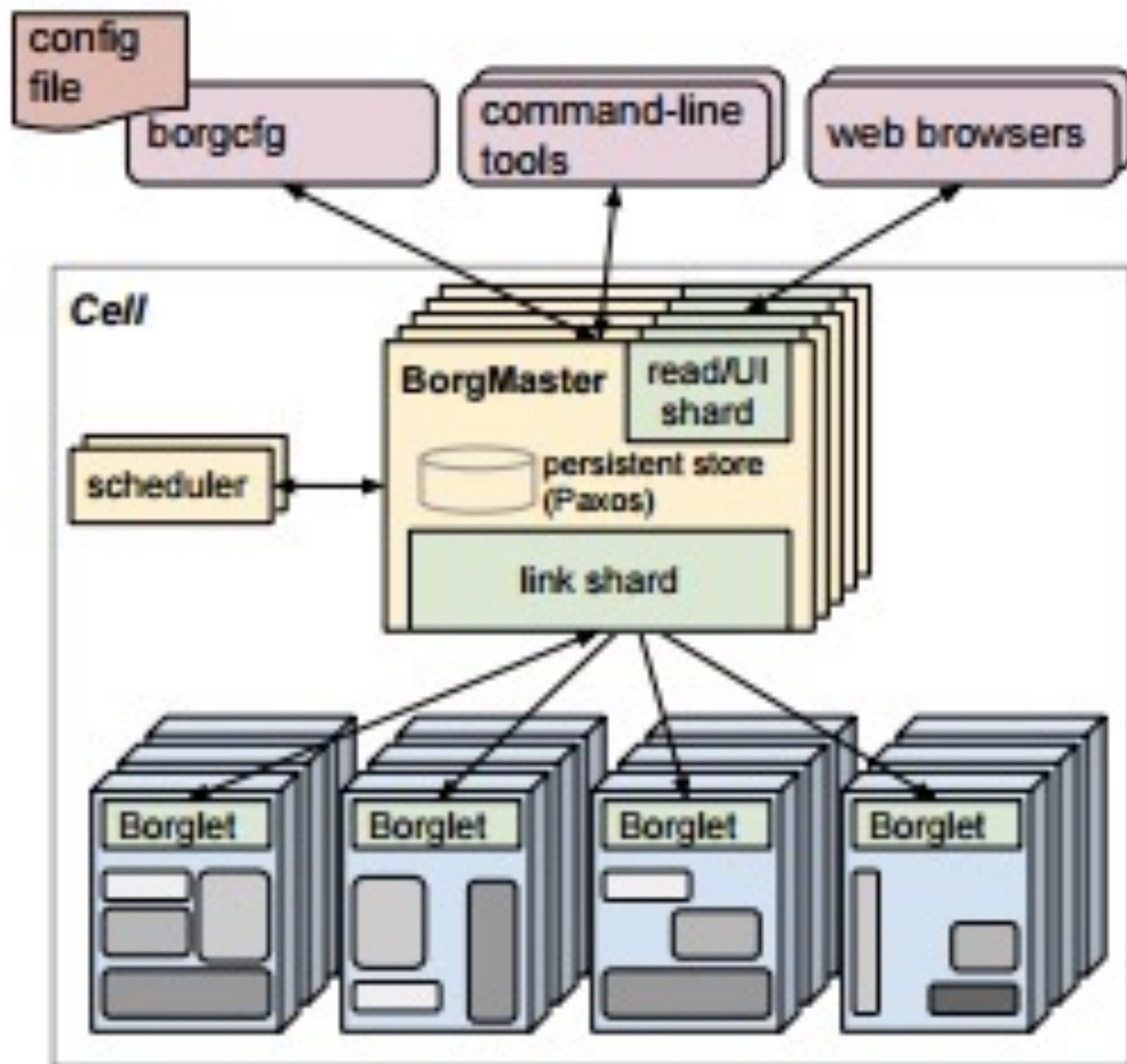
Google Borg

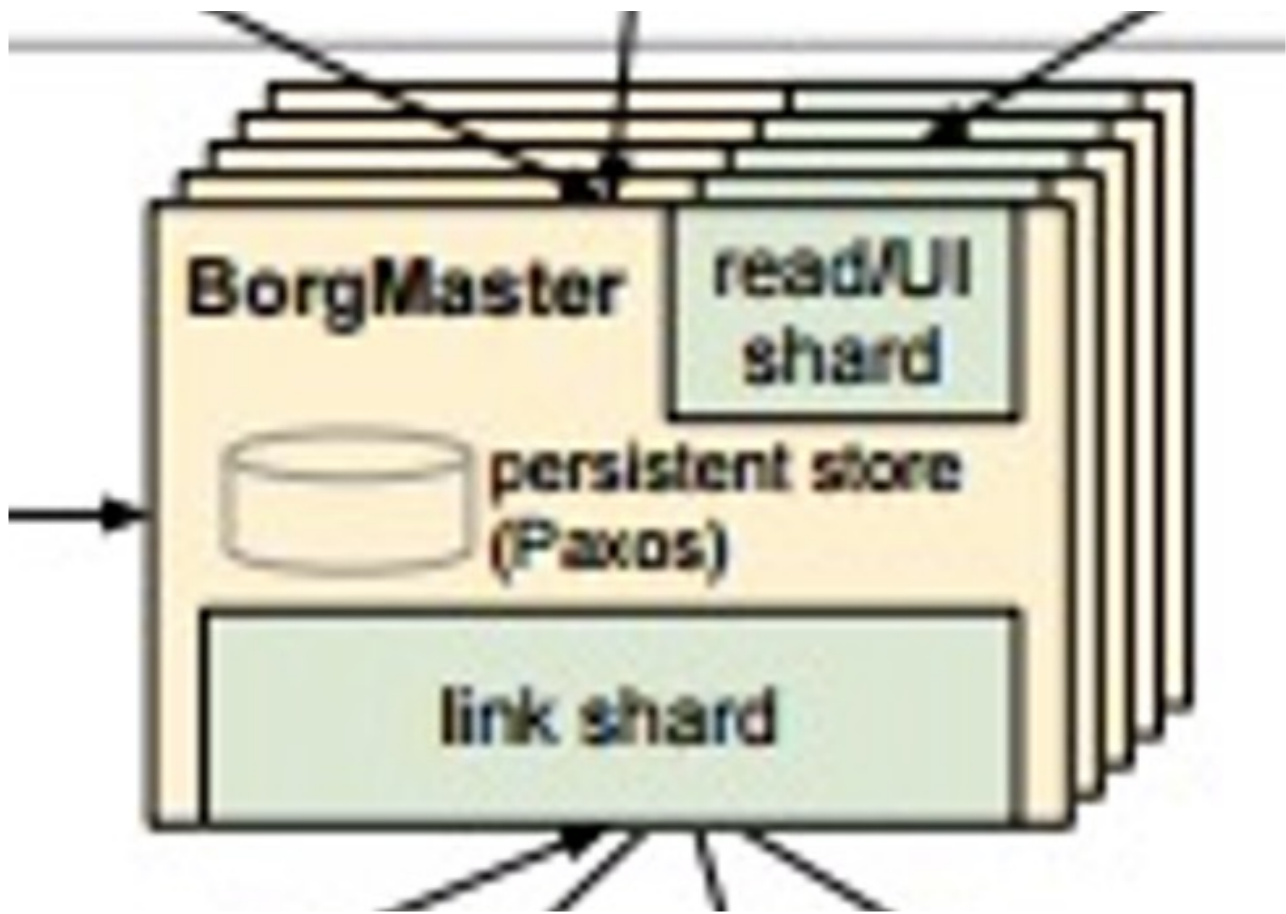


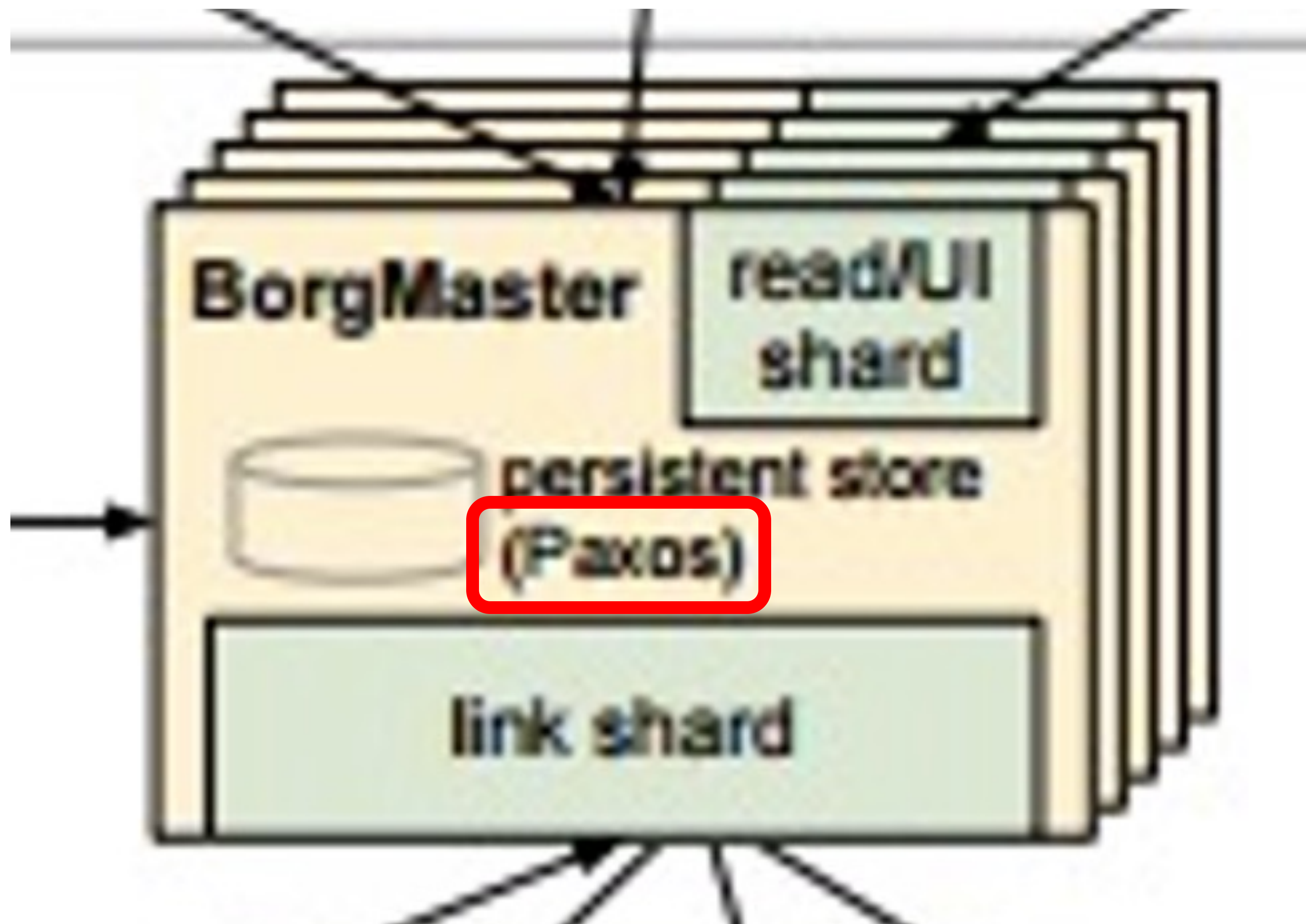
Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines

数十万のジョブ
数千のアプリケーション
数万のマシン

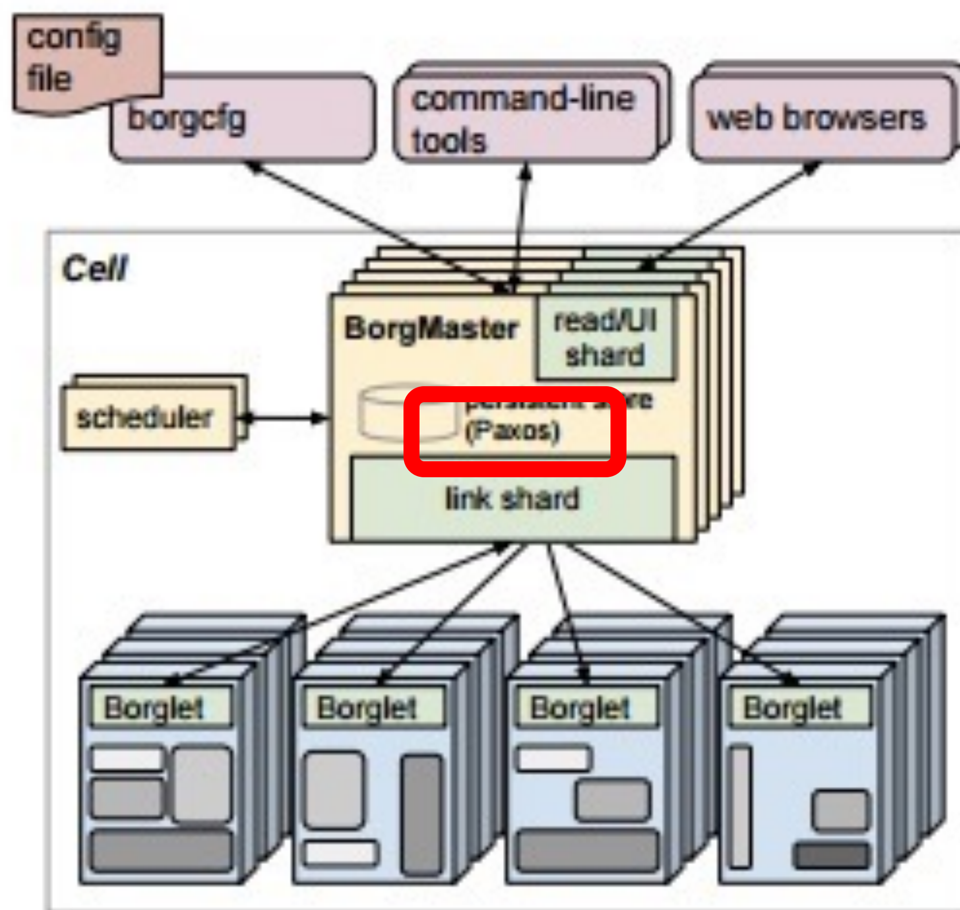

config file
borgcfg
command-line tools
web browsers
Cell
BorgMaster
read/UI shard
persistent store (Paxos)
scheduler
link shard
Borglet
Borglet
Borglet
Borglet


config file
borgcfg
command-line tools
web browsers
Cell
BorgMaster
read/UI shard
scheduler
persistent store (Paxos)
link shard
Borglet
Borglet
Borglet
Borglet


BorgMaster
read/UI
shard
persistent store
(Paxos)
link shard


BorgMaster
read/UI shard
persistent store
(Paxos)
link shard

Google Borg

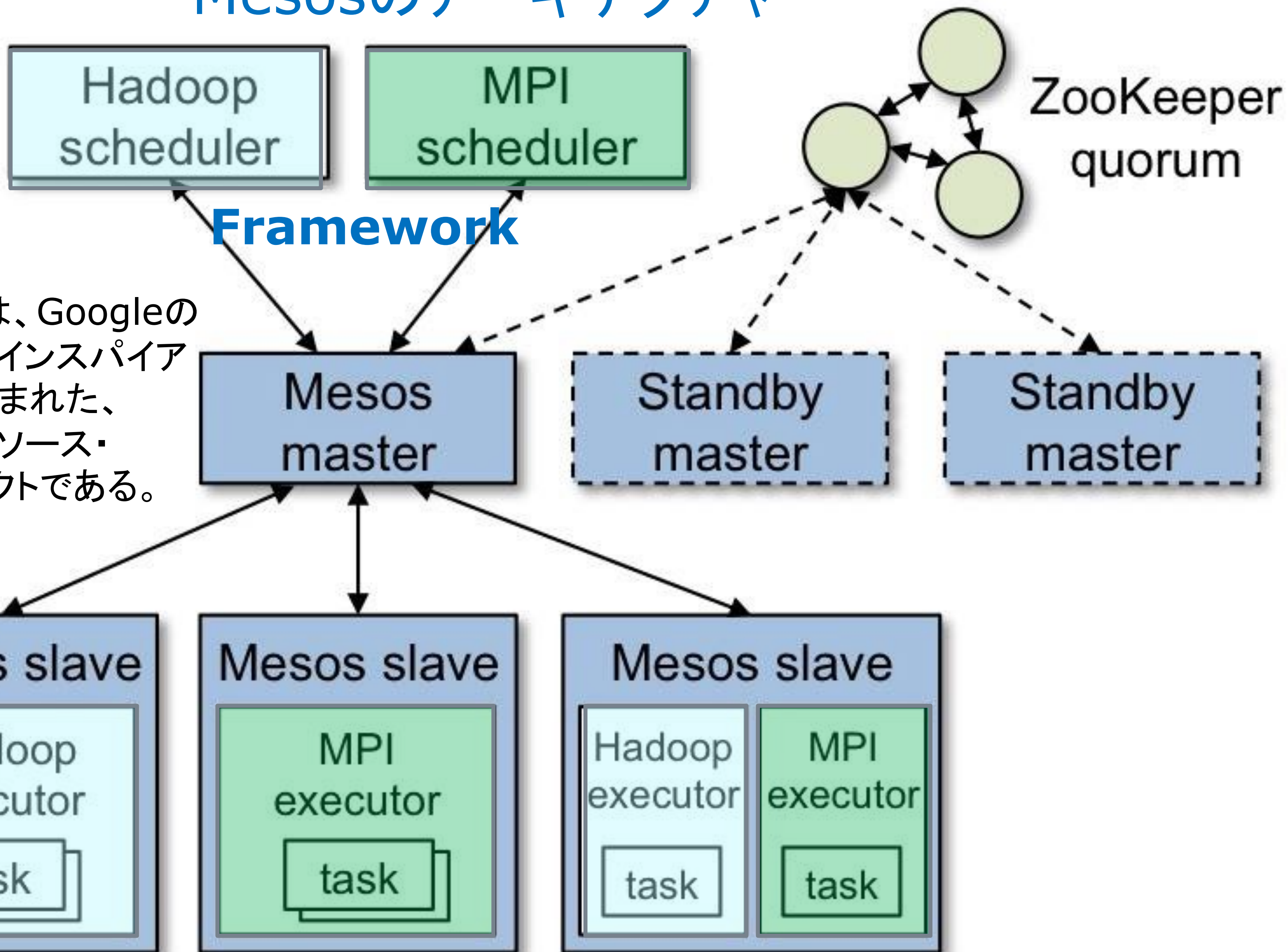


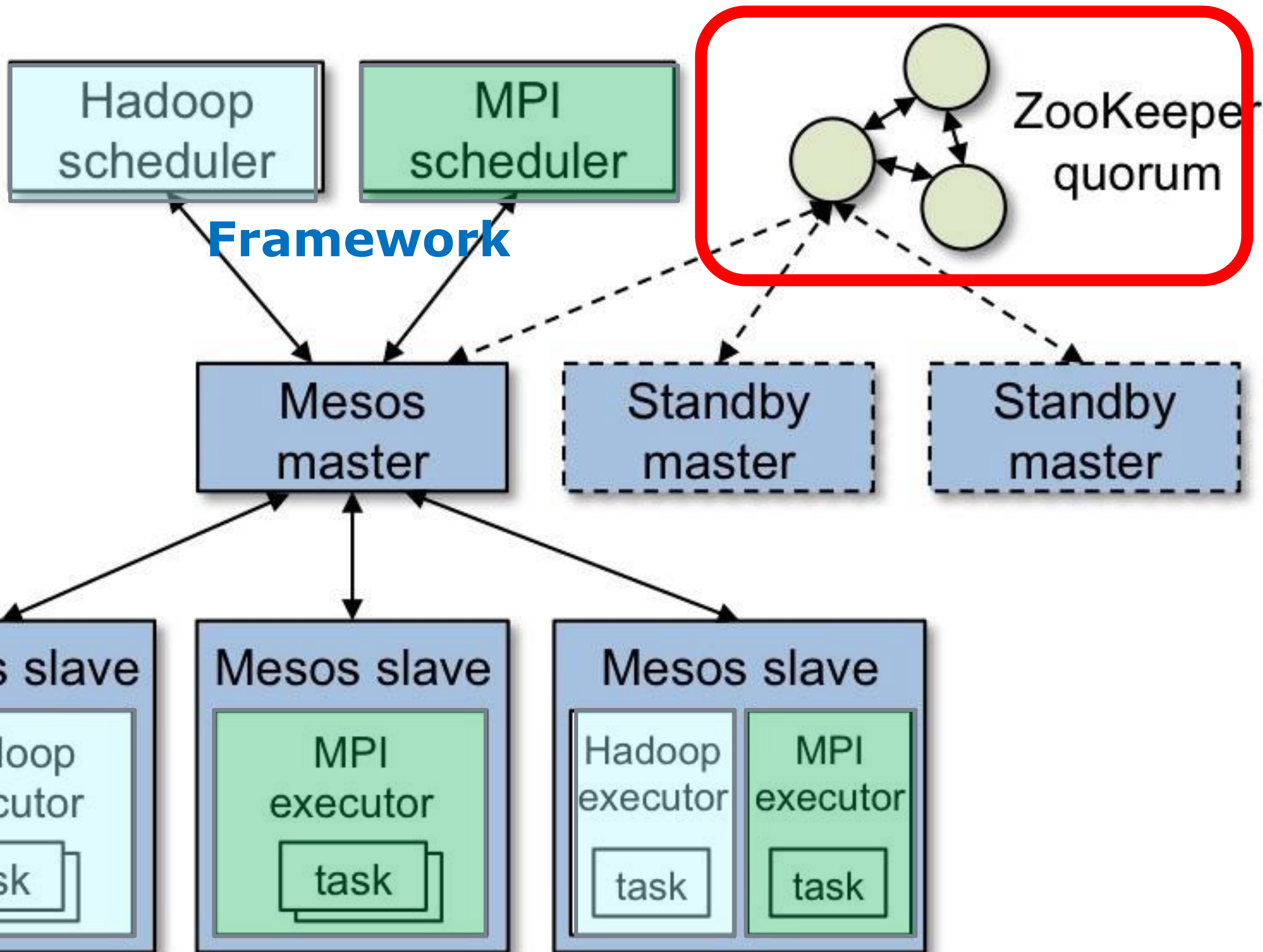
Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines

数十万のジョブ
数千のアプリケーション
数万のマシン

Mesosのアーキテクチャー

Mesosは、Googleの**Borg**にインスパイアされて生まれた、オープンソース・プロジェクトである。




Hadoop scheduler
MPI scheduler
Framework
ZooKeeper quorum
Mesos master
Standby master
Standby master
Mesos slave
Hadoop executor
task
Mesos slave
MPI executor
task
Mesos slave
Hadoop executor
task
MPI executor
task

LeaseとPaxos

マスター・ノード

ノードに障害が起きた時、そのノードの切り離し・新しいノードの補充を自動的に行うことは、大規模分散システムにとって重要な課題である。

分散システムでは、システム全体の統合・調整を担う、マスター・ノードと呼ばれるノードが特別な役割を果たす。

どの時点でも、システムの中心となるマスター・ノードは、必ず存在しなければならない。しかも、マスターは、一つだけである。

障害を引き金としたノードの再構成が行われたとしても、二つのノードが、同時にマスターになってはいけない。

マスター・ノードが２つ？

マスター・ノードの復旧は、単なるレプリカ・ノードの復旧とは異なる問題がある。

マスター・ノード Aが、何らかの理由で、ノード Bと接続できなかったとしよう。

Bの側から見れば、A との接続がうまくいかないことは、マスターノードが死んだように見えるということである。Bは自分がマスターになることを提案する。

他のノードがそれに同意したとして、そうした提案がなされていることを A が知らなかったとすれば、Aは自分がマスターであるかのように振る舞うだろう。

マスターの交代にLeaseを利用する

この問題に、次のように対応することができる。

マスターが死んだ時（死んだと思われる時）、あるノードは、自分がマスターになることを提案する。

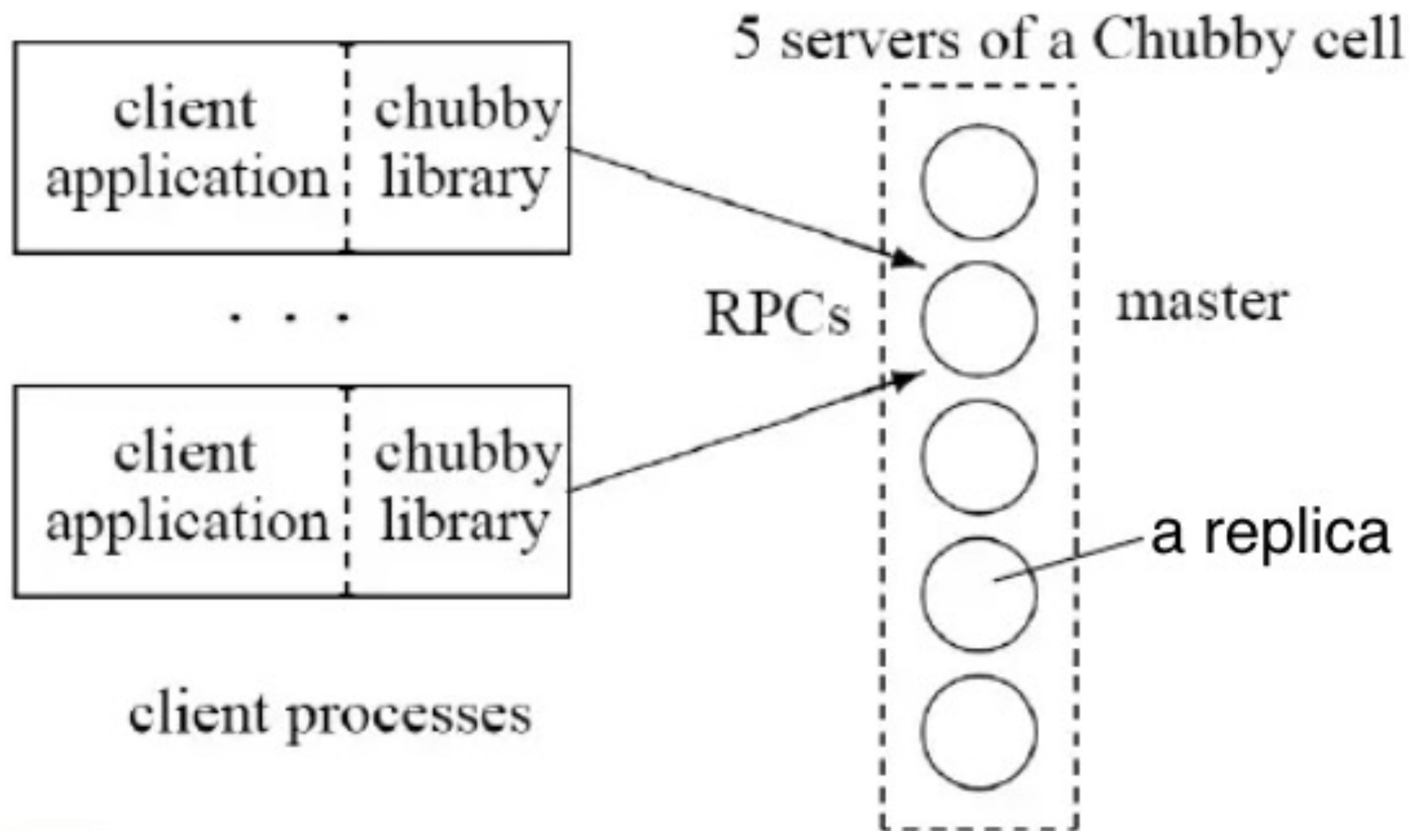
他のノードがこの提案を受け取った時、彼らがこの提案を受けいるのは、旧マスターのリースが切れている場合だけである。

提案を受け取ったノードの多数が、それを受け入れた時、はじめてマスターの交代が起きる。

あるノードがマスターになった時、それはマスターのリースの期間中だけ、マスターであることができる。

マスターは、ノードの多数が受け入れた時、マスターのリースの期間を延長できる。

GFS, BigTableで利用された ロック・サービス -- Chubby



第三部

Paxosアルゴリズム

第三部の目標と参考文献

Paxosのアルゴリズムの解説です。

- Leslie Lamport
  ”The Part-Time Parliament”
  https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
- Lamport
  “Paxos Made Simple”
  https://lamport.azurewebsites.net/pubs/paxos-simple.pdf
- Jim Gray and Leslie Lamport
  ”Consensus on Transaction Commit”
  https://arxiv.org/pdf/cs/0408036.pdf

第三部

Paxosアルゴリズム

1. たとえ話で理解するPaxos (1)
2. たとえ話で理解するPaxos (2)
3. Paxos アルゴリズム (1)
4. Paxos アルゴリズム (2)

たとえ話で理解するPaxos (1)
By ”The Part-Time Parliament”

“The Part-Time Parliament” 論文

Paxosは、エーゲ海のギリシャの島の名前である。

ある考古学者が古代ギリシャのPaxos島で奇妙な議会制度が発達していたことを発見し、その意思決定のスタイルが現代の分散システムの設計にも役に立つのではという論文を発表する。それが、“The Part-Time Parliament”という論文である。

論文発表の経緯 – 編集者から

この投稿は、最近になって、編集部のキャビネットの中から発見されたものだ。古いものだが、編集長は掲載する価値があると考えた。著者は現在、ギリシャの島々でフィールドワークをしていて連絡が取れないため、私が掲載の準備の依頼を受けた。

著者は考古学者で、コンピュータサイエンスにはほとんど興味がないようだ。また、残念なことに、彼が紹介した謎に包まれた古代Paxos文明は、ほとんどのコンピュータ科学者にとって興味のないものだろう。

にもかかわらず、その立法システムは、非同期環境での分散型コンピュータシステムの実装方法の優れたモデルとなっている。実際、Paxos人がプロトコルに加えた改良点のいくつかは、コンピュータ・システムの文献では知られていない。

たとえ話としての
“The Part-Time Parliament” 論文

もちろん、この論文を書いたのは、Leslie Lamport 本人なのだが。

ここでは、この架空の物語を書いたLamportが、Paxos島の議会制度に想定したものをまずみておこう。それは、アルゴリズムとしてのPaxosの、とてもすぐれた「たとえ話」による解説になっている。

L. Lamport. The part-time parliament. ACM Trans. on Computer Systems, 16(2):133–169, May 1998.
https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf

古代Paxosの議会

議会の主な任務は、国の法律を決定することである。それは議会が通過させた一連の法令によって定義される。

現代の議会では、その行動を記録するために書記を雇う。ただ、Paxosでは、議会の会期中ずっと議場に残って書記を務める人はいなかった。その代わりに、Paxosの議員は、自分の行動を記録するため台帳を持ち、可決された法令の番号順に記録していた。

台帳の整合性

議会プロトコルの最初の要件は、台帳の一貫性だった。

つまり、2つの台帳に矛盾した情報が含まれてはならないということだ。

法令が最終的に議会を通過して元帳に記録されることを保証するためには、何らかの要件が必要であった。

彼らの一箇所にとどまらず歩き回る性癖が問題だった。

ある議員グループが法令を可決した後、宴会に出かけた後、別の議員が議場に入り、何も知らずに矛盾した政令を可決したとしたら、整合性が失われる。

議事の進行

十分な数の議員が議場に留まらない限り、議事の進行は保証されない。Paxosの議員は議会外での活動を抑えようとしないため、どんな法令も確実に成立させることができなかった。

しかし、議員たちは、会議場にいる間は、自分とその仲間が、議会に関するすべての事柄について迅速に行動することを保証した。この保証があったからこそPaxosでは、次のような議事進行条件を満たす議会プロトコルを考案することができた。

議員の過半数が会議場にいて、十分に長い時間、誰も議場に出入りしなかった場合は、議場にいる議員が提案したあらゆる法令が可決され、可決された法令は、議場内のすべての議員の台帳に記載される。

議員とメッセンジャー（使者）

議場の音響環境は悪く、議員同士の口頭での会話は難しかった。議員たちは、使者によってのみコミュニケーションをとることができ、必要なだけの使者を雇うための資金が与えられていた。

使者は、メッセージを間違えないようにすることはできても、伝えたことを忘れてしまったり、同じメッセージを、再度伝えてしまうこともあった。彼らは、自分が仕える議員と同じように、時間の一部だけを議会の仕事に費やしている。使者は、何か別の仕事をするために会議場を離れることもある。その場合、メッセージは届かないことになる。

議員や使者はいつでも議場を出たり入ったりできたが、議場内では議会の仕事に専念していた。議場にいる間、使者はタイムリーにメッセージを届け、議員は受け取ったメッセージに迅速に対応していた。

たとえ話で理解するPaxos (2)
By ”The Part-Time Parliament”

投票と定足数（Quorum）

宗教会議の法令は、番号付きの投票用紙を使って採決される、一回の投票では、一つの政令について、同じ番号の投票用紙が用いられる。投票で聖職者は、政令に賛成するか、投票しないかのいずれかを選ぶことしかできない。

投票には、投票が成立するための定足数がある。その数は、聖職者の集合に対応している。投票が成立し法案が可決されるのは、定足数に含まれるすべての聖職者が法案に賛成した場合だけである。

投票を構成するもの

投票Bは、形式的には、次のような構成要素からなる。(特に断りのない限り，集合は有限集合を意味するものとする）。

- B_{dec}　法令（投票の対象）
- B_{qrm}　聖職者の空でない集合（投票の定足数）
- B_{vot}　聖職者の集合(法令に賛成票を投じた人たち)
- B_{bal}　投票用紙の番号

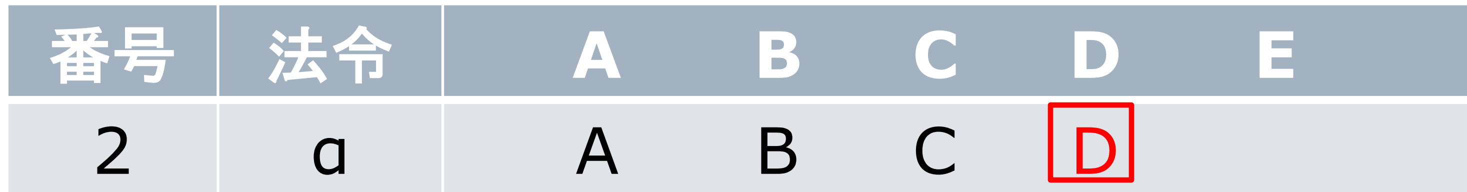
投票の数学的性質

Paxosの数学者は、投票の集合**B**に3つの条件を定義して、行われた投票の集合がこれらの条件を満たしていれば、投票の一貫性が保証され、議事の進行が可能であることを示した。

- **B1(B)**　**B**の中の投票は、全てユニークな投票番号を持つ。
- **B2(B)**　**B**の中の任意の二つの投票のquorumには、少なくとも一人が、共通に含まれる。
- **B3(B)**　**B**の中の投票Bは、**B**の以前の投票で、quorum内のだれかが賛成票を投じていれば、Bの投票の対象である法令は以前の投票の中で最新の法令に等しい。

投票セットの例と先の三つの条件

Quorum　赤字は投票者

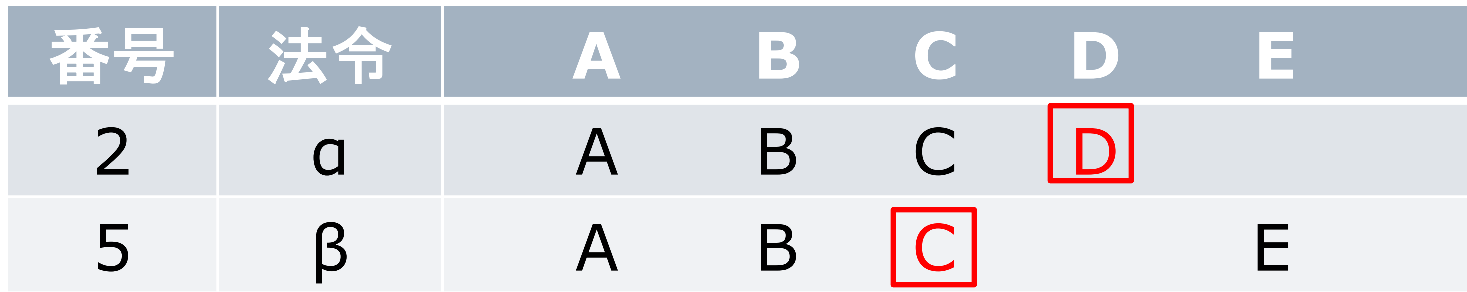


番号	法令	A	B	C	D	E
2	α	A	B	C	D	

番号2の投票は、最も最初の投票である。投票についての条件は満たされている。

投票セットの例と先の三つの条件

Quorum　赤字は投票者



番号	法令	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E

番号2の投票は、最も最初の投票である。投票についての条件は満たされている。

番号5の投票では、4人のquorumメンバーは、誰も、以前の投票には参加していない。投票についての条件は満たされている。

投票セットの例と先の三つの条件

Quorum　赤字は投票者

番号	法令	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E
14	α		B		D	E

番号14の投票では、以前の投票に参加したことのあるquorumメンバーはDだけである。Dは、投票番号2の投票に投票している。

先の条件**B3**は、投票番号14の等票の対象の法令は、投票番号2の法令と同じことを要求する。

投票セットの例と先の三つの条件

Quorum　赤字は投票者

番号	法令	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E
14	α		B		D	E
27	β	A		C	D	

番号27の投票は、投票が成立している(quorumメンバー全員が、賛成表を投じている)Aは以前の投票には、参加していない。Cが参加した以前の投票は投票番号5の投票、Dが参加した以前の投票は投票番号2の投票である。一番最近のは、投票番号5である。

先の条件**B3**は、投票番号27の等票の対象の法令は、投票番号5の法令と同じことを要求する。

投票セットの例と先の三つの条件

Quorum　赤字は投票者

番号	法令	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E
14	α		B		D	E
27	β	A		C	D	
29	β		B	C	D	

番号27の投票のquorumメンバーは、B, C, Dである。この中で、Bは以前14の投票に、Cは27と5の投票に、Dは27と2の投票に参加している。これらで一番最近のものは、27番の投票である。

先の条件**B3**は、投票番号29の等票の対象の法令は、投票番号27の法令と同じことを要求する。

Paxos アルゴリズム (1)
By “Paxos Made Simple”

Paxosでのノードの三つの役割

Paxosでは、ネットワーク上のノードは、つぎの三つの役割を持つ。

- Proposers （提案者）
- Acceptors （受容者）
- Learners　（学習者）

ノードは、一つ以上のクライアントの役割を持つ。
（通常は、一つのノードが、上記3つの役割を果たす）

Phase 1 (prepare)

PaxosアルゴリズムのPhase 1は、合意の為の準備である。proposerはacceptorに投票への参加を呼びかける。

- proposerは、提案する番号 nを選んで、acceptorの多数に、nを含んだ、prepare リクエストを送る。
- acceptorは、番号 nを含んだprepare リクエストを受け取って、もし、番号 nが、今まで見たどのprepareリクエストに含まれた番号より大きければ、このリクエストにYESのレスポンスを返す。この時、番号がnより小さい提案は受け付けないことを約束し、それまでに受け取った（もしもあれば）、最も大きい番号をレスポンスに含めて送る。

Phase 2 (Accept)

PaxosアルゴリズムのPhase 2では、proposerが提案した値での合意を受け入れるよう acceptorに呼びかける。

- もし、proposerが、そのprepare リクエストに対してacceptorの多数からYESのレスポンスを受け取ったなら、proposerはそれらのacceptorの一人ひとりにacceptリクエストを送る。このacceptリクエストには、提案した番号nと値vが含まれていて、その番号nは、受け取ったレスポンスの中で最も大きい番号を持つ提案の番号の値である。

Phase 2 (Accept)
提案の可決 -- 値の選択

- acceptorは、番号nをもつ提案に対するacceptリクエストを受け取ったなら、もしも、nより大きな番号を持つprepareリクエストに既に答えていないならば、その提案を受け入れる。

番号nを持つ提案の値vは、acceptorの多数が、phase 2で提案された番号の値を受け入れた場合にのみ、選択される。

Phase 3 (Learn)

合意した値vは、すべてのLearnerに送られる。

いくつかのオプションがある

- それぞれのacceptorは、提案を受け入れたら、それを全てのlearnerに送る。
- acceptorは、特別なlearner（大抵はproposer）にそれを送る。特別なlearnerは、その結果をブロードキャストする。

Paxosの性質

P1: 全ての提案の番号は、ユニークである。

P2: acceptorの任意の二つの集合は、少なくとも一人のacceptor を共通に持つ。

P3: phase 2で送られる値は、phase 1 の全てのレスポンスで最も大きい番号を持つ提案の値である。

P3 の解釈

番号	値	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E
14	α		B		D	E
27	β	A		C	D	
29	β		B	C	D	

Paxos アルゴリズム (2)
By ”Consensus on Transaction Commit”

Paxosとは何か？

Jim Gray と Leslie Lamportの論文
“Consensus on Transaction Commit” のAbstract から

分散トランザクションのコミットの問題では、トランザクションをコミットするか中止するかについて合意に達する必要がある。従来のTwo-Phase Commitプロトコルは、コーディネーターに障害が発生した場合にブロックする。

耐障害性を持つ合意アルゴリズムは合意に達するが、プロセスの大部分が機能しているときは決してブロックしない。

Paxos Commitアルゴリズムは、各参加者のコミット/中止の決定に対してPaxos合意アルゴリズムを実行して、$2F + 1$個のコーディネーターを使用するTransaction Commitプロトコルを取得し、少なくともそれらの$F + 1$個が正常に機能して場合には進行する。

Paxos Commitは、Two-Phase Commitと同じように、安定したストレージへの書き込み遅延があり、障害のない場合に同じメッセージ遅延を持つように実装できるが、より多くのメッセージを使用する。

従来のTwo-Phase Commitアルゴリズムは、Paxos Commitアルゴリズムの$F = 0$の特別なケースとして取得される。

The Paxos Consensus Algorithm

コンセンサスの問題では、アクセプターと呼ばれる一連のプロセスが値を選択するために連携する。各アクセプターは異なるノードで実行される。基本的な安全性の要件は、単一の値のみを選択することである。トリビアルな解決策を除外するには、選択した値がクライアントによって提案された値でなければならないという追加の要件がある。生存性の要件は、アクセプターのノードの十分な大きさのサブネットワークが十分長い間障害がない場合、最終的に何らかの値が選択されることを主張している。厳密な同期の仮定を示さなくても、F個のアクセプターが失敗したとしても、コンセンサスを達成するには2F+1のアクセプターが必要であることを示すことができる 。

Paxosアルゴリズムは、ポピュラーな非同期コンセンサス･アルゴリズムである。これは、非負の整数で番号付けされた一連の投票番号を使用し、それぞれにリーダーと呼ばれる事前に定義されたコーディネーター･プロセスがある。投票番号0のリーダーは、初期リーダーと呼ばれる。通常の障害のないケースでは、最初のリーダーが提案された合意の値を受け取ると、すべてのアクセプターにこの値と投票番号0を含むPhase2aメッセージを送信する（Phase1については以下で説明する）。各アクセプターはこのメッセージを受信し、投票番号0のPhase2bメッセージで応答する。リーダーが大多数のアクセプターからこれらのPhase2bメッセージを受信すると、値が選択されたことを通知するPhase3メッセージを送信する。

最初のリーダーが失敗し、バロット0が値を選択しない可能性がある。その場合、新しいリーダーを選択するためにいくつかのアルゴリズムが実行される。たとえば、Aguilera et al. のアルゴリズムである。 単一のユニークなリーダーを選択することは、コンセンサス問題を解決することと同じである。ただし、Paxosは一貫性を維持しており、複数のプロセスがリーダーであると考えている場合でも、2つの異なる値を選択することはできない。（これは、複数のコーディネーターが不整合につながる可能性がある従来の3Phaseコミットプロトコルとは異なる。）単一の障害のないリーダーは、生存性を確保するためにのみ必要である。

自分を新たに選出されたリーダーであると信じるプロセスが投票を開始して次の段階に進行する。（複数のリーダーが存在する可能性があるため、複数のPhaseのアクションが同時に実行される場合がある。）

Phase 1a

Phase1aは、proposer(leader)からacceptorへのリクエストで、ある投票番号で合意形成のための会議をするので参加してほしいという会議参加の呼びかけである。当然、全てのacceptorに送られる。送られるのは、投票番号のみである。prepareリクエストともいう。

このprepareリクエストに対して、acceptorは、この会議に参加するなら、phase1bレスポンスをproposer(leader)に返す。このphase1bレスポンセをpromiseレスポンスともいう。

ある場合には、acceptorは、このpromiseレスポンスを返さない。

Phase 1a

リーダーは投票番号*bal*を選択する。番号*bal*はリーダーのものであり、Phase 1が行われる間のどの投票番号よりも大きいと信じられる。リーダーはすべてのアクセプターに、投票番号*bal*をPhase1aメッセージで送信する。

Phase 1b

phase1bは、proposer（leader）からのprepareリクエストによる会議参加の呼びかけ（phase1a）に対する、accepterからの、私は会議に参加しますという約束の返信である。promiseレスポンスとも呼ばれる。

acceptorが、このpromiseレスポンスを返すのは、proposerがprepareリクエストで提示した投票番号が、新しいものだった（これまでの投票番号より大きい）場合である。

逆に、今回、proposerが提示した投票番号より大きな投票番号の提示を既に受けていたとすれば、acceptorは、proposerの誘いにpromiseレスポンスを返さない。

Phase 1b

こうしたケースは、異なる投票番号を提示するproposerが二人いれば起こりうる。リーダーが二人いるようにも見えるのだが、Paxosは、そうした想定を排除していない。二つの会議が召集されたら、その会議に紐づけられた投票番号が大きい方を選ぶと考えていい。

prepareリクエストには、投票番号しか含まれていなかったが、それぞれのacceptorは、promiseレスポンスに、次の情報を追加して、proposerに返す。

- これまでprepareリクエストで提示された最大の投票番号
- これまで参加した会議で提案に賛成票を投じた最も最近の投票番号

Phase 1b

アクセプターは投票番号のためのPhase 1aのメッセージを受信する。もし、投票番号*bal*、*または、それより大きい値bal*のためのアクションがそれまで行われていなかった場合、アクセプターは、次のような現在の情報を含んだPhase 1b メッセージで応答する。

- Phase 1aメッセージで受信した最大の投票番号、および
- 送信した投票番号が最も大きいPhase2bメッセージ（存在する場合）。

アクセプターは、投票番号balあるいはそれより大きいbalに対してのアクションが既に行われていれば、Phase 1aメッセージを無視する。

Phase 1 から Phase 2へ

phase 1では、phase1aのprepareリクエストでも、phase1bのpromiseレスポンスでも、やり取りされるのは投票番号のみで、内容的には、「会議に参加してください。」「はい。参加します。」ということを確認しているだけである。

phase1では、proposer(leader)は、実は合意されるべき提案を提案していないのである。

phase2に入って、初めて、proposer(leader)による合意内容の提案（phase2a）と、acceptorたちによる、合意内容の採決（phase2b）が行われる。

Phase 2a

phase2aで、proposer(leader)は、合意すべき内容を、全てのacceptor たちに提案する。この内容を合意として受け入れてほしいという呼びかけで、acceptリクエストと言われる。

phase2aで、最も重要なプロセスは、proposer(leader)が、どの様にして合意すべき提案を作るかということである。それには、phase1で、promiseレスポンスを通じてacceptorたちから集めた情報を使う。

Phase 2a

二つの可能性がある。

一つは、もし、promiseレスポンスを返したacceptorが一人も以前に合意の採択に加わったことがない場合である。この場合、proposer(leader)は、任意の提案を行うことができる。

番号	提案	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E

番号2の投票は、最にとっても最初の投票である。proposer Dは、任意の提案を自由に行うことが許される。Dはαを選択した。

番号5の投票では、4人のquorumメンバーは、誰も、以前の投票には参加していない。proposer Cは、自由に、提案βを提案した。

ただ、どちらの提案も、その時点では、採択されていない。合意はまだない。

Phase 2a

もう一つの可能性は、acceptorのうちの何人かが、採決に加わったことがある場合、それらのうち、もっとも新しい採決の対象となった提案vを、proposerは改めて合意の対象として提案する。

番号	提案	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C	↑	E
14	α		B		D	E

番号14の投票では、以前の投票に参加したことのあるquorumメンバーはDだけである。Dは、投票番号2の投票に投票している。

proposer B または Dは （この表だけでは、どちらがproposerであるかは分からないのだが）、αを提案する。

この段階でも、まだ、合意は得られない。

Phase 2a

これは、以前のセクションで、Paxosの性質としてあげた、B3,P3に基づく選択である。

こうした選択をするのに必要な情報は、すべて、acceptorからproposerに寄せられたpromiseレスポンス（phase1b）の中に含まれている。

Phase 2a

リーダーが、投票番号balのPhase1bメッセージを、大多数のアクセプターから受信していれば、リーダーは 次の2つの可能性のいずれかを学習できる。

- **Free** アクセプターの多数派の誰もPhase2bメッセージを送信したと報告しないなら、アルゴリズムはまだ値を選択していない。
- **Forced** 多数派の一部のアクセプターは、Phase2bメッセージを送信したと報告しているとする。μが 報告されたすべてのPhase 2bメッセージの最大投票番号としよう。M_μを投票番号μを持つすべてのPhase 2bメッセージの集合とする。M_μの全てのメッセージは、すでに選択されたかもしれない、同一の値vをもっている。

Freeの場合、リーダーは任意の値を受け入れようとすることができる。通常、クライアントによって提案された最初の値を選択する。
Forcedの場合、全てのアクセプターに送った値vと投票番号 bal を持つPhase 2aメッセージから、値vを選ぼうとする。

Phase 2b

phase2bは、「この内容を合意として受け入れてほしい」というproposerからのacceptリクエスト(phase1b)に対するacceptorからの返信 accept レスポンスである。

acceptorは、proposerからの acceptリクエストの投票番号が、いままでacceptorが扱ってきた投票番号より大きいものだったら、acceptレスポンスを返す。提案の内容を吟味する訳ではない。

acceptorがacceptレスポンスを返すということは、acceptorがproposerの提案に賛成しますという賛成票を投ずることである。

Phase 2b

proposerは、acceptorからのacceptレスポンスが多数を占めていれば、提案が受け入れられ可決されたことを知る。合意は京成された。

番号	法令	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E
14	α		B		D	E
27	β	A		C	D	

いったん合意が形成されれば、proposer(leader)は、その結果を全ての人に伝える新しいフェーズ phase3 に入る。

Phase 2b

番号nの提案で値vが選ばれたとすれば、その後のphase 2の提案で送られるどんな値もまたvでなければならない。

番号	提案	A	B	C	D	E
2	α	A	B	C	D	
5	β	A	B	C		E
14	α		B		D	E
27	β	A		C	D	
29	β		B	C	D	

Phase 2b

アクセプターが、値がvで投票番号 balのPhase 2aメッセージを受信した場合。もし、より大きい投票番号のPhase 1a または Phase 2aのメッセージをまだ受け取っていないなら、アクセプターは、そのメッセージを受け入れ、値*v* と投票番号balを持つ phase 2b メッセージを リーダーに送る。アクセプターは、すでにより大きい投票番号に参加していれば、そのメッセージを無視する。

Phase 3

phase3は、phase2で proposerとacceptorたちが形成した合意をLeanerたちに伝えるフェーズである。

Two-Phase Comitで言えば、commitを行なうことに対応している。

phase3が終わると、proposer(leader)は、新しい投票番号を用意して、再び、phase 1に戻る。

Phase 3

リーダーは、アクセプターの多数から、値vと投票番号balのPhase 2bメッセージを受信すると、値vが選ばれたことを知り、この事実を全ての関連するプロセスに、phase 3メッセージで伝える

安全性(Safety)と生存性(Liveness)

- 安全性
  - 提案された値だけが選ばれる
  - 一つの値だけが選ばれる
  - ノードは、実際に値を選んだ後に、初めてその値が選ばれたことを知る。
- 生存性
  - ある提案された値が、最終的には選ばれる。
  - もしも、ある値が選ばれれば、ノードは最終的には、その値を知ることができる。